Multidimensional Ordered Mappings for Empirical Machine Learning Research

Lorenz Weizsäcker, Johannes Fürnkranz
{lorenz,juffi}@ke.tu-darmstadt.de

Abstract

We propose a plain data cube type that covers machine learning experiments through a run model build upon it and trough functions associated with the type itself. As container type it is agnostic to the domain. This leads to an experimentation carcase that the user has to fill from scratch but can save efforts on diverse projects at low total orientation costs.

1 Introduction

In the area of experimental machine learning there are various frameworks that allow the experimenters to employ approved, high level tools for divers purposes. Roughly speaking, there are two types of framework incentives: the *engine* type wants to solve a specific type of task. In order to employ it the user is expected to build framing code by which the provided engine is called. The *embracing* incentive wants that experiments are run through the given framework which subsequently calls engines that can be plugged-in in terms of wrappers.

Although writing embracing code is mostly straightforward, it can be error-prune and time consuming as well and it therefore is plausible to share such code too. However, defining an embracing framework is rather ambitious as it somehow has to formalize the intentions of potential users with respect to the experimental setup and the employed data structures. A trade-off between comfort versus generality is inevitable: a narrow specification on what experiments are allows broad pre-build services for the specified type of experiments but it also make the framework less likely to be applicable to the different projects of the users.

In this work we discuss an embracing framework that sets the trade-off much in favor of generality. At its core it basically consists of a data cube type, a cascade of ordered multidimensional arrays, holding the (intermediate) results of the computations. The run of an experiments is modeled along this type.

The container is agnostic to the type of content and the definitions of data source types, processing units, evaluation schemes and so one are not part of the model. The model is a skeleton along which we can distribute the experimental code and provides a general reference system by means of a name space. The named entities are called e-nodes and correspond to dimensions of the cube mentioned above. An experiment is defined as tree of e-nodes such that any part of the experimental code need to be covered by an e-node and thereby becomes a cube dimension. This means that many of those dimension will have a single value only and it is a characteristics of the model that it does not distinguish between such singleton dimensions and multi-valued parameters.

A rich source for machine learning software is [2]. Even when restrict to python you can find several embracing python frameworks such as MLPY [1], PyMVPA [7], scikit-learn [13], or plearn [16]. They have in common that they provide many training algorithms, pre-processing, and evaluation features such that the user freely can combine those in order to build the desired experiments. Beside the engines they also provide top-level code that implements typical experimentation pattern like gird-search. These can be used smoothly because that embracing code is aware about the interfaces of the engine shipped within the same framework. To this respect the proposed model cannot compete, the user has to come up with appropriate code units and interfaces herself. On the other hand the definition of new problems types or intermediate steps is much of what machine learning research is about for that ready made interfaces can be too concrete.

Also, we like to encourage user to incorporate foreign, but specific and well tested libraries like Nonnegative Matrix Factorizion [19], OpenOpt [8], libDAI [12], chi2 kernel [9; 10], Pyriel [5] and at the same time relieve those tools from providing top level code. By stronger abstraction in the top level system, the user has indeed to care for engines, data types, and interfaces but at least some system expertise remain valid even when task changes from, say, word-sense disambiguation to change point detection. This is important since the cost-gain ratio of a framework becomes better when we can apply it for different occasions. A similar spirit is followed in [3] where also a conservative modeling is proposed but there in conjunction to java.

In the next section we define the central data cube type and subsequently explain the experiment model in detail. Section 3 is dedicated to services that are associated with the data cube and can be used to deal with (intermediate) results and services based on the experiment model. The section much refers to the prototype implementation [18; 17] you can consult for further illustration. In the last section discuss a view aspects of the approach, and point to selected features of the implementation that we became fond of.

2 Experiment Model

An experiment is defined as an directed ordered tree of *experimental node*s or *e-node*s which represent dimensions of the experiment such as *stemming routine*, *data source* or a numeric parameter. The tree sets the order in which the nodes are deployed. It thereby has to respect the dependencies among nodes as given by their definitions.

When a node is deployed it produces a sequence of outputs that are taken as alternative values we want to probe on the respective experimental dimension. A node is deployed for any combination of output values of the nodes that it takes as inputs. And so may serve its outputs one by one as input for other nodes in turn.

The output values of a node are stored in a data-cube, called *v-cube*. It holds a multidimensional array that has one dimension for each node the values directly or indirectly depend on and recursively comprises cube for these dependencies.

We first give more details on v-cubes. They hold the results with respect to one e-node in the experimental tree. Then, we cover the definition of e-nodes and thereby explain how the experimental tree, in short *e-tree*, defines the flow of deployment of its nodes.

2.1 V-Cubes

A v-cube has a finite number of *dimensions* that have a unique *name* each. As they stems from an experiment compounds, the dimensions of the v-cube come along with an acyclic *dependency relation* as well as a *linear order* that is consistent with the dependencies: a dimension a precedent to a dimension b in the linear order must not depend on b.

The last dimension is called *target dimension* and this is the one with which the *values* that v-cube holds in a multidimensional *array* are associated. The dimension of the array correspond one-to-one to the cube dimensions. The values of the non-target dimensions . That is why v-cube is a recursive type: each dimension of a v-cube is associated with exactly one v-cube where the target-dimension is associated with the v-cube at hand. We refer to the non-target dimension also as *parent dimensions* and the associated v-cubes as *parent v-cubes*, and we speak of the *main v-cube* when we want to refer to a v-cube excluding its parent cube. For each parent dimension there is exactly one parent v-cubes with that dimension as target dimension. However, a v-cube may have no parents at all. Still, the data array has at least one dimension since the target-dimension is always present. Figure 1 gives an illustration.

The number of values the array can hold along a dimension is referred to as *size* of the corresponding v-cube dimension. The sizes have to be consistent: a dimension shared with a parent cube must have have the same size in the parent cube and the cube at hand. Given the size of a dimension we can index the values along that dimension and we assign one *value-label* to each index of the target dimension. Any entry of the array can be identified by a tuple dimension-name–index pairs. Since parent v-cubes refer to a subset of the dimension at hand, we can associate each entry with a unique *parent entry* of the array of any given parent v-cube.

In view of our application we allow array entries to be non-existing or *void*. However, there is again a consistency constraint: a void entry can only be parent to void entries.

2.2 E-Nodes

An experiment is defined as an ordered tree of e-nodes. First let us look at linear e-trees only, e.g. experiments that are defined as a sequence of e-nodes, as in the example given in figure 2. Such sequences directly corresponds to a v-cube associated with the last node and we demand the nodes to have *name*s that are unique within the sequence.

Descent Production
Each e-node has a *descent function* that produces a list of one or more *descent values* of arbitrary type. The function comes with a set of *descent input names* that refer to ancestor *input nodes* (if there are branches, two or more e-nodes may have identical names, we explain in 2.5 how the references are resolved in this case). The descent function is called for each combination of descent value produced by the input e-nodes. For given call we refer these as *current values*. The descent values of an e-node are stored in its *descent cube*, a v-cube. The target dimension of the cube is given by the name of the e-node and the parent v-cubes are the descent cube of the input nodes or of e-node on which the input nodes depend in turn. The dependencies are derived from the descent input names and we explain in subsection 2.5 how exactly the dependencies are collected. The current values are the parent values of the outputs of the descent production stored in the in the descent cube.

The number of descent values, the so-called *descent size* may depend on the input values. This can make the entries for the upper indexes in the target dimension void. The size of the target dimension of the attached cube is given by the maximal descent size.

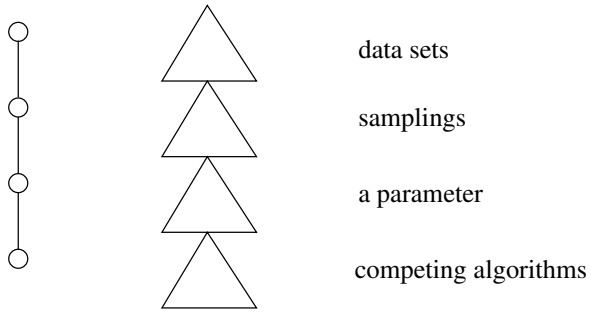

Figure 2: A linear e-tree with four nodes (left). Each produces several values per descent call, e.g. per combination of descent values of the ancestors that are taken as inputs. After the run we have a tree of descent value (middle) that are stored in v-cubes. See also figure 1.

Ascent Production
After the descent production of the node and its possible descendants is completed, a node can additionally produce so-called *ascent value*. Opposed to descent the *ascent function* produces a single value only. Likewise descent, the ascent function is called for each combination of ancestor descent values and it therefor can take ancestor node as inputs. But it further can depend on successor nodes. In cases where the referred successor node defines an ascent function as well, the input refers to its ascent value. Otherwise all its descent values produced under the current values of the referring node are passed as an v-cube.

The ascent value is stored in the *ascent cube* another v-cube that is associated with the e-node if it defines an ascent function. For reasons that become clear in section 2.3, ascent functions are less important than it it might appear on first sight.

Sibling Values
The descent and ascent production is separately done for each combination of ancestor descent values. In order to compute outputs iteratively we may want to access the previous outputs of the same node, the so-called *sibling values*. Let a be ancestor of b. Then b can access its proper

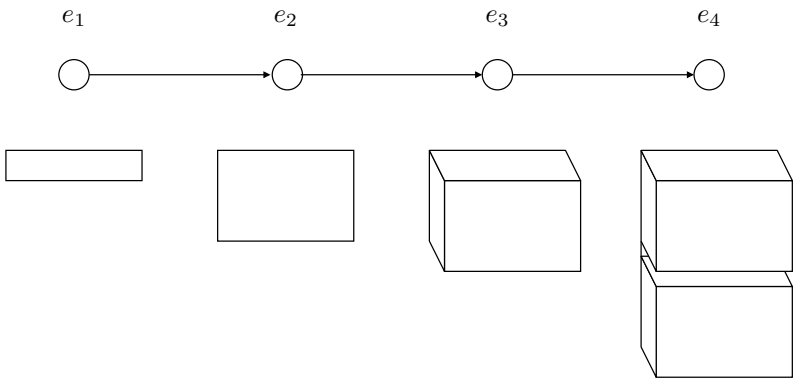

Figure 1: Top: four e-nodes. An outgoing edge indicate that the nodes output is input for the other node. Below: Each node is primarily associated with one v-cube. The target-dimension, the v-cube itself, and the respective e-node have the same key, for instance e_4. For each non-target dimensions the v-cube e_4 references the respective ancestor e-node/v-cube. The v-cube e_4 hence comprises all cubes above it, thereby holding a cascade of arrays with increasing number of dimensions.

outputs that it made under the previous descent value of a. To this end, the production function of b simply refers to b itself as input node. Since b may have other ancestors we further need to specify the *sibling pivot* a when we actually access a sibling value. Also, we need a function that provides *initial sibling values* for the case that b produces under the first value of a.

2.3 Non-Linear E-Trees

Within a linear e-tree, an e-node b that depends on another e-node a considers one descent value of a at a time. If you want access all descent values of a, for instance for aggregation purpose, you need a different buildup where a is deployed before b but not as direct ancestor but in a preceding branch in the e-tree, see figure 3. The e-tree is deployed in depth-left-first order. Assume a node has two children a and b where a comes first. For given descent values of the common ancestors of a and b, the subtree rooted in a is deployed first. Then, it is turn to the subtree with head b. From this subtree we can access the all descent values of a and its descendants that where produced under the given decent values of the common ancestors. This can be use for aggregation over the produced values, for instance, we may build the mean over descent values of a, or select an optimal parameter with respect to a certain output.

Preceding branches, e.g. subtrees that are non-ancestor to an node e are not reflected as dependencies of production of e. One reason is that they always have a single realization given the common ancestors likewise the ascent function returns a single value only.

2.4 Advanced Tree Handling

Pruning

In case the realization of an e-node and its successors is, for certain input values, logically unwanted or technically impossible, we can pass over the deployment of the node for these input values. To this end the node defines a *prune-condition* that can take the same inputs as the descent function, or a subset thereof, as arguments and outputs whether the tree of values should be pruned at this node. The prune-condition is evaluated before the descent-function is called, e.g. the *pruning* is applied above the node. Unless the prune-condition holds for all inputs, the not computed entries do exists in the value cube but they remain void as it the case when the node has varying descent size.

Looping

E-Nodes may also be used to represent compounds of an algorithm. Algorithms may contain loops where the number of iterations depends on the result of the loops body. So far e-nodes may have varying number of descent values depending on ancestor values that are possible inputs. But since successors, which correspond to a loop body, cannot be accessed as descent inputs, the descent size cannot depend their values. The reason for this restriction is that the descent-function is called before the successors a deployed.

In order to allow *indefinite descent sizes* we can make the node to be deployed repeatedly. The new descent values are appended to those computed so far while the ascent value is overwritten with each iteration. The node defines a *loop-condition* that should output a non-false value until no further iteration is wanted. It can take the same inputs as the ascent functions, including successor nodes that are passed as v-cubes, and it is called right after the ascent-function. For instance, you may check whether the value of a leaf belonging to the current iteration has changed compared the previous one.

2.5 Completing the Model

Above, we let out some details on the experiment model which we now want to catch up. First, we want to fix to what exactly the input names refer depending on the type of production an the position of the node in the tree. Second, we state how the dimension of the v-cubes associated with an e-node are set. Finally, we clear the proceeding in cases where an e-node has an ancestor on which it does not depend.

Reference Resolution

Let us define a view notions on the relative positions between nodes of an ordered tree. Consider the linear order on all nodes that results from depth-first traversal of the tree respecting the linear orders on siblings. For a given node a this order divides the nodes of the tree into three groups: a itself, the ancestors of a, and the successors of a. More specifically we define the following relations a node b can have to the node a.

- *t-ancestor*: b is on a path from the root to a, including root, excluding a itself
- *s-ancestor*: in depth-first order b comes before a but b is not a t-ancestor of a

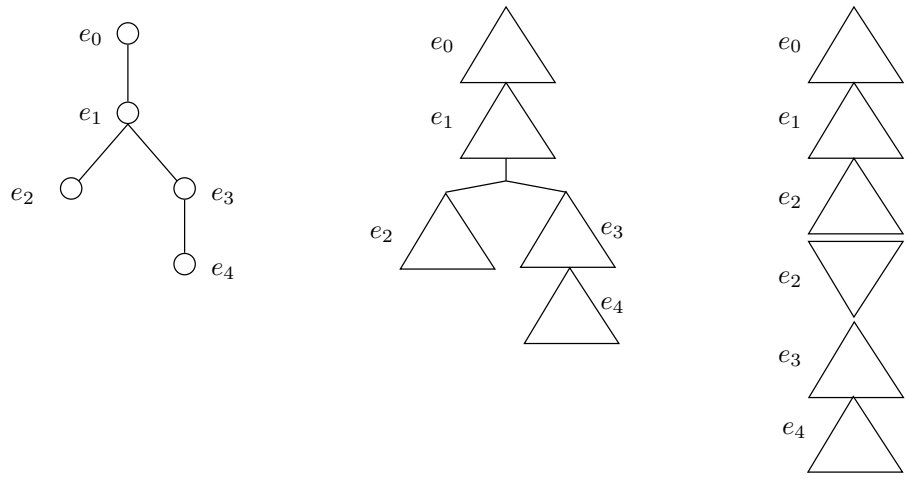

Figure 3: A non-linear e-tree with e-nodes $e_0, \ldots, e_5$. Since e_2 precedes e_3 in depth-left-first traversal of the tree, e_3 can access all descent values of e_2 at a time at least those that are build based on the current descent values of the common ancestors. Alternatively, e_2 can digest the descent values of itself and of possible node below itself by means of ascent production. In either case the result of the subtree under e_2 logically is one value and can be accessed as such by the nodes e_3 and e_4 as sketched on the right.

- *self*: b is a
- *t-successor*: a is t-ancestor of b
- *r-successor*: all other nodes

The r-successors are the nodes that a cannot access and that is why we do not distinguish further subgroups.

The nodes are linked together via the input names of their production functions. The first thing to fix is to what node a given argument name refers to in case there are two or more nodes with identical names. The node is selected as follows.

- Self first.
- Ancestors go before successors.
- Within these two groups the one that comes last in the depth-first order is chosen.

Per e-node there is only one mapping assigning nodes from the e-tree to input names. It applies to all production function of an e-node. We actually define *production functions* as those function of an e-node that understand the argument names as names of e-node and resolve them in the way just explained. They further have in common that they all influence the descent or ascent values of a node. Beside descent and ascent production, they include prune-condition, loop-condition, and initial-sibling-values and a view more that have have not mentioned here but can be found in the class definition of e-nodes in the implementation.

Since a node b can have descent and ascent values we have to fix which of both is passed to a node a given the relation of b to a.

- self: descent function gets a descent value and ascent function an ascent value.
- t-ancestors: a always gets the current descent value.
- s-ancestor or t-successor: a gets the ascent-values in case b defines an ascent function. Otherwise, a gets the descent-values of b. In either case only those values are passed that are produced for current descent values of the common t-ancestors.

Dependency Resolution

Before we deploy the e-tree we have to fix the dimension of the descent and ascent cube associated with the nodes. Each dimension of a cube represents an e-node on the values of which the outputs of the production function and thereby the entries of the cube depend on. For this reason we refer to the dimensions here also as *dependencies* of the cube. Following the above reference resolution a dependency refers to descent values or to ascent values of the referred node, but only to either of them. Further, we do not distinguish dependencies of the descent versus the ascent cube because this gives little savings but resolution the scheme would become a bit opaque.

As a start, we state that the cubes depend on the e-node at hand it-self. For the dependency resolution we basically have to take the *indirect dependencies* in the wright order into account. To this end a single depth-first traversal suffices. The production function are partitioned into *pre-* and *post*-functions depending on whether they are called before or after the deployment of the child-nodes. The latter group can access t-successors, while the former cannot. If a node e has a dependency a that is used by a pre-function of e, the dependencies of a are added to those of e before the children of e are visited. In case the dependency is due to a post-function, the dependencies of a are added when the children of e are done.

Referential Transparency

We apply *referential transparency* in the following sense. When the descent function is to be called a second time under the same combination of *indexes* of the ancestor descent values, the call is skipped, since the values for that inputs has already been computed. This occurs whenever there is node e that has an ancestor a that produces multiple descend values but on which e does not depend. By catching this the dependencies are reflected in production calls. In general referential transparency is a bit delicate notion, as is refers to *value equality* of inputs and outputs of a function, [15]. Here, we refer to the cube-indexes, what makes the concept quiet simple, as it is clear to what equality we refer to.

```
[ data(),
  sample(),
  kernel(),
  kernel_tray(),
  [[ c(),
     sub_sample(n_folds=6),
     algorithm(),
     leaf( input_refs =
           [['sample', 'sub_sample']])
   ] ,
   [ c_opt(),
     algorithm( input_refs =
                [['c', 'c_opt']]),
     leaf()
   ]
  ]
]
```

Figure 4: An e-tree with one branching. The e-nodes in the tree are instances of e-node classes such as defined in the code snippet in figure 5. The `input_ref` arguments replace input names, such that e-node definition are applicable in either branch.

```
class c(ENode):

  def descent(self):

    return range(-4,4)


class c_opt(ENode):

  def descent(self, leaf):

    vc_opt = leaf.aggg([
      ('argmax', 'c'),
      ('mean','sub_sample'),
      ('sel', 'leaf', 'accuracy')
                       ])

    return [ vc_opt.get_value() ]


class algorithm(ENode):

  def descent(self, data, c,
              kernel, kernel_tray):

    engine = LIBSVM(
      data_tray = data,
      kernel    = kernel,
      C         = 2.**c,
      kt        = kernel_tray,
                  )

     return [ engine ]
```

Figure 5: Definitions of e-nodes. The node `c_opt` shows the digestion of an s-ancestor by means of v-cube-aggregation. In `algorithm` some library is called.

3 Services

We coded a python prototype named *peewit* [18] that implements the experiment model defined in the previous section and provides various services. Some are for dealing with v-cubes, others, which we refer to as *housekeeping services*, are to support the experimental work-flow. Some services of the latter type only partially rely on the model and could be implemented to some extend within any other top-level framework. Still, they fit the spirit of this work as they show that we can provide helpful code independent of domain types and interfaces. For more detailed information on the services see [17].

3.1 V-Cube Services

V-cube is a sort of data cube [6], a notion for multidimensional container types used in the fields of Online Analytical Processing [4], [14]. As such we can implement various OLAP operations. The presence of sufficient data cube functionality in terms of data manipulation and presentation is crucial for the experimental framework. Due to limited resources we can only realize a fraction of the services that we would like to use. But some are implemented by date and they can illustrate the benefits of cube services: presentation in plain text or as plots, manipulation like selection, aggregation, or merging.

The remaining subsection has three parts. First, we describe concrete services for v-cube manipulation. The last one described, namely merging, leads to the problem of v-cube alignment that we discuss separately. Last, explain how pre-computed v-cubes can be integrated into an experiment.

V-Cube Manipulation

Likewise other data-cubes v-cubes come with several routines of cube manipulation. The most simple ones are the following.

- *value selection*: select or delete a slice given as index or value-label for some dimension
- *permutation of dimension*: change the order on the dimension
- *permutation of values*: for a given dimension change the order on value-labels and the corresponding slices

The manipulations have to account for the main cube as well as for the parent cubes. Thats is why the next, apparently also simple manipulation, is, in fact, logically a bit more complicated.

- *plain aggregation*: summarize the values along the aggregated dimension into a single value

It will occur that that a parent cube also has the aggregated dimension but the aggregation cannot be applied to its values. Then these values becomes void. The aggregated dimension will be shrunk to a single dummy value. As an example think of taking the mean. This cannot be applied to the parent cube if the parent cube is non-numeric. And even if it were numeric, it would not always make sense to apply mean to it just because we queried the mean values on a node that has the cube as parent.

For aggregations such $\min$ or $\max$ it makes sense to ask for the/an *fulfilling object* that leads to the result value, for $\min$ and $\max$ such aggregation are known as argmax and argmin.

- *argument aggregation*: for a set of ancestors dimensions get a combination of values such that the target

value holds some condition with respect to all value combinations of those ancestors

Since the main cube comes along its parent cube peewit can easily provide this kind of aggregation as well. The user specifies the *aggregants* that are the dimension over which the aggregation is to be done. The result is a v-cube with has those dimensions of the main cube as proper dimension that are not specified as aggregants. If the user specifies multiple aggregants, the result one v-cube per aggregant.

Another direction of manipulation is *cube merging*.

- *merging*: make a single v-cube out of two

There can be different levels of service here that are defined by different capabilities in matching the dimension and values of the cubes to be merged. At a lower service level, called *straight merge*, the service demands that the cubes are identical in dimension and sizes except the sizes of the dimension along which the cube are to be merged.

V-Cube Alignment

We may also demand more advanced merging that is robust against non-identical names/labels of dimensions/values and tries to solve a matching problem. Assume we are given two cubes v and $\bar{v}$ that may differ in dimension and values. An *v-cube alignment* consists of *matching* of the dimension of v to those of $\bar{v}$ and, for each matched pairs of dimension, of a matching of the indexes of the respective dimensions. Whether the matching are restricted to be total or not depends on the application of the alignment. For instance, with merging partial dimension-matching (in either direction) might be wanted whereas the index-matchings must be total except for the merge dimension.

Certain cases that appear in practice are likely matchable in automatic manner: the index order of a given dimension has changed, the name of a dimension has changed, or in one cube there are additional dimension but with a single index only. For solving the alignment we can analyze name and label information but also compare the content of the cubes, likewise for ontology matching problems.

Descent Values from V-Cubes

Since v-cubes correspond to linear sub-trees we can also apply them as such. Assume you are about defining an e-tree for an experiment that shares dimensions with an v-cube you have at hand. You can add the v-cube like a node into the raw e-tree and when the e-tree is read, this node will be expanded to a linear subtree.

The number of nodes in this sub-tree depends on the ancestors of the v-cube in the raw e-tree. When peewit finds that a dimension of the cube is present as ancestor, this ancestor will be identified with the dimension of the e-node. If no fitting ancestor can be found the dimension and the corresponding parent v-cube translates into an e-node that is included in the e-tree. In the first case peewit also tries to match the descent values of the ancestor node with the values from the parent v-cube such that this linking of v-cubes is likely to work even even if the ancestor produces a smaller number of values than present in the included cube or if the order of values differs.

3.2 Housekeeping Services

Persistent Name Space

By writing experimental code the user introduces names for compounds and values. Peewit extends the live time of the names in the sense that the user can use the names for further coding as well for reading, processing and, presentation of the results. The names of the dimensions of an v-cube are given by the e-name of the node they stem from. For a given dimension, the value labels that are assigned to the positions an entry can take along that dimension or are provided by the user or derived from the values at that position.

The e-names, aka dimension names, are used as follows.

- The names of the arguments of production functions must be e-names. The e-name arguments refer to node in the tree and thereby define the dependencies between nodes.
- This also means that the node names are used in the body of the production function.
- The names of dimension are used in textual or graphic representations of cubes. This way we can easily find the production functions that are responsible for an element in the representation.
- By means of the run-numbers these links remain valid even if the production functions have been changed since the v-cube was produced.

As the concept of value labels is part of the v-cube type the application of label always goes through v-cubes. Value labels correspond to row or column headers in tables and allow identification and selection of rows and columns by names. This can be used to handle results but also in the experimental code itself since v-cube are build during run and can be accessed during run by means of preceding branches or ascent production. Peewit tries to define the labels automatically by means of an heuristic, see [17], but the user can also define the labels explicitly.

Regime on Paths

The way peewit handles paths has three aspects. First, peewit encourages *user space coverage* meaning that all data and code files are accessed within home directory of the user and linked by the project configuration. This facilitates the portability of the files in the sense that it can be synchronized more easily between different machines and allows self-contained project snapshots.

Second, the system uses a simple scheme for naming files and placing them into flat ensemble of directories. Peewit produces several output files such as serialized v-cubes, log-files keeping the standard output, plots, and tex-snippets. The concrete path scheme may be disputable but main profit is that there *some* reasonable scheme that is transparent for the user and readable to the system at the same time.

Archiving

The archiving service aims a more effective persistency of the code and other data that defines an experiments. To this end we employ *version management systems* and add a thin layer upon them which allows us to recover former stage of the covered files in a comfortable way:

- each experiment run gets a *run-number*, a sort of on-top revision number
- the run-number is attached to results and, at low-key, displayed in result representations
- with that number at hand you can query peewit to restore the files as they were when the result was produced

Peewit does not conflict with existing version management instances but rather allows their usage at different levels of integration: no archiving, archiving aside possibly existing repos, or archiving that integrates existing repositories. We particularly recommend the service. It does not cause additional work for the user and still gives her a quick access to the code that were used to produce former results. Further, the project can be exported such that other researchers can access past states of the code to comprehend how the results shown for instance in a particular plot were obtained. It presumes, however, that you are ready to disclose your work in such a deep way.

Parallelization
Consider a node e the t-successors of which do not refer sibling values with respect to e or any of its ancestors. In this case the deployment of the subtrees under the different descent values of e are concurrent such that peewit can distribute the computations without further coding provisions.

Once the cluster is setup by means of about ten configuration values it suffices to define the *launch axes* that are the e-nodes over which the computation is to be parallelized. Peewit then cares for distributing the jobs and assembling the partial results and finally returns a single v-cube to the user as if the experiment were run on the local machine.

3.3 Proposed Services

Below are two more housekeeping services that are rather clear on a conceptual level but have not yet been implemented for peewit.

Voluntary Type Checking
Peewit is based on python which is a dynamically typed language. This means that the production function do not have a signature that would explicitly force inputs and output to be of specific types. We do want not step into the *dynamic versus static typing* discussion [11] here, but propose a mechanism for *voluntary type checking* at run-time as follows.

The user can, but does not have to, define e-node methods for type checking, one for each input nodes and one for each production function that make checks on the outputs. The checks are not restricted to test class memberships but can test arbitrary conditions. This means that the user freely decides on whether she wants to check in-depth, superficially, or not at all. If the user defines checking, the checking code is separated from body of the production functions and thereby is out of the way. We might also tell peewit to omit the checks in situations that are not suited for fixing interface agreements.

Progress Indication
When a run takes more time, it is convenient to have an idea what fraction of the computation is done and, ideally, how long the remaining productions will take. This kind of service clearly lies in domain of a top-level experimentation framework and the experiment model of peewit seems well suited for progress indication because we can, for instance, easily count the number of the right-most leaf descent values that have been produced so far.

Unfortunately, it is not that simple to get the total number of descent values before the run it completed since the number of decent values an e-node produces can vary within one run. Though in many cases this number is fixed for all node in the tree it still can be difficult to get that number in advance even having the abstract syntax tree at hand. A simple solution is to provide a progress indication that takes nodes with presently unknown or varying size by means of an unknown factor into account, for instance stating that 65x out of 320x rightmost leafs have been produced.

4 Discussion

We start this section by pointing to some conceptional issues, then pick features of the framework that we found particularly helpful when doing experiments with it, and finally give a short summarizing conclusion.

4.1 Issues

Data Cubes
With the v-cube we basically repeat accomplished work on OLAP data cubes. Likewise SQL for tables there are serveral implementations, though mostly commercial, of cube arithmetics. It therefore would be reasonable to build upon this work. We decided to define a proper cube type because we are particularly interested in a smooth interplay of the cube type with the run model and want to have the option of adapting the type to the needs of experiments. The drawback is that the data cube functionality is to be implemented from scratch including tools for actually viewing the cubes content. We are still locking for a better solution to this issue.

Robust Access
In general we belief in cube types for keeping (intermediate) result machine learning of machine learning experiments. In particular, we think that the particular structure of v-cubes is suited for automatically reintegrating an v-cube into an e-tree or merging and combining it with other v-cubes. The presence of names *and* instances over different dimension facilitate solving automatic matching of the indexes. This is important to make cube combination robust against smaller differences in names or content that would not foreclose a reasonable (partial) combination of content but prohibits simple one-to-one matching by name or index identity. However, until we have not brought automatic matching to certain level we cannot try it and it remains unclear to what extend one can profit from matching in practice.

Additional Code Layer
The decomposition into e-nodes adds some top-level structure to the experimental code. By naming the e-nodes you some-how add a logical layer that describes the aspect of an experiments and their dependencies. The name of e-node would express the purpose of the covered code for the experiment while the names of the included library typically refer to what the authors of the library think is its main application. However, this is not a solution to the problem of keeping your libraries in order. On the contrary, covering the code below e-nodes inhibits the view to the bottom and may detain you from building a neat structure in your code.

4.2 Highlights of the Prototype

Experiment Outline
The user provides the final structure of an experiment by stating the e-tree in *branch notation* that particularly readable if the tree has a view branches only as it is the case for most experiments. This explicit tree representation also serves as outline of the experiment much like an index of contents of an article or book. It does not disclose the full information on the semantics and dependencies of the nodes but with speaking node names, likewise the section

titles in the table of contents, you get a good idea on the design of the experiment.

Run Numbers
The archiving service costs small delay when starting a run and we only used the restore function a couple of times. But it is very agreeable that we do not have to make notes on each run because we can look up the details on some dimension by passing the run-number attached to a plot to the restore function and looking for the definition of the respective e-node.

Grid Computing
Parallelization of fully concurrent computations is logically straightforward. Still, in practice you have to solve a bundle of issues: formulation of the job-assignments, transferring file to and from the cluster, and merging the partial results. We still experienced trouble when the synchronization broke but in sum the parallelization with peewit pleasantly works under the hood.

4.3 Conclusion

We proposed a plain container type to hold values produced during an experiment that depend on several dimension and we defined an tree model on experiments where each branch correspond to a cube of that type. The model is abstract and ignores any kind of machine learning knowledge. But it can be used to factorize the experiment into its compounds and thereby get a grip on its structure. The prototype implementation illustrates that such a structure is sufficient for several helpful services that are valid independent of the concrete type of problem under study. We understand this work as a draft in the search of structures for machine learning and other computational experiments that allow sharing experimentation code based on a low common denominator.

References

[1] D. Albanese, R. Visintainer, S. Merler, S. Riccadonna, G. Jurman, and C. Furlanello. mlpy: Machine learning python, 2012.

[2] M. Braun, S. Sonnenburg, and C. S. Ong. machine leanring open source software, 2007. `http://mloss.org/`.

[3] R. E. de Castilho and I. Gurevych. A lightweight framework for reproducible parameter sweeping in retrieval. In C. T. Maristella Agosti, Nicola Ferro, editor, *Proceedings of the 2011 workshop on Data infrastructurEs for supporting information retrieval evaluation*, DESIRE '11, pages 7–10, New York, NY, USA, Oct 2011. ACM.

[4] P. M. Deshp, J. F. Naughton, K. Ramasamy, A. Shukla, K. Tufte, Y. Zhao, D. Shasha, D. B. Lomet, D. Barbara, and M. Franklin. Data engineering, 1997. Warning: the year was guessed out of the URL.

[5] T. Fawcett. PRIE: a system for generating rulelists to maximize ROC performance. *Data Min. Knowl. Discov*, 17(2):207–224, 2008.

[6] J. Gray, S. Chaudhuri, A. Bosworth, A. Layman, D. Reichart, M. Venkatrao, F. Pellow, and H. Pirahesh. Data cube: A relational aggregation operator generalizing group-by, cross-tab, and sub-totals. *Data Mining and Knowledge Discovery*, 1(1):29–54, Mar. 1997.

[7] M. Hanke, Y. Halchenko, P. Sederberg, S. Hanson, J. Haxby, and S. Pollmann. Pymvpa: a python toolbox for multivariate pattern analysis of fmri data. *Neuroinformatics*, 7:37–53, 2009. 10.1007/s12021-008-9041-y.

[8] D. Kroshko. Openopt, 2009. `http://mloss.org/software/view/55/`.

[9] C. Lampert. chi2 kernel, 2009. `http://mloss.org/software/view/64/`.

[10] C. H. Lampert, M. B. Blaschko, and T. Hofmann. Beyond sliding windows: Object localization by efficient subwindow search. In *CVPR*, pages 1–8, 2008.

[11] E. Meijer and P. Drayton. Static typing where possible, dynamic typing when needed: The end of the cold war between programming languages, 2004. `http://research.microsoft.com/en-us/um/people/emeijer/Papers/RDL04Meijer.pdf`.

[12] J. M. Mooij. libDAI: A free and open source C++ library for discrete approximate inference in graphical models. *Journal of Machine Learning Research*, 11:2169–2173, Aug. 2010.

[13] F. Pedregosa, G. Varoquaux, A. Gramfort, V. Michel, B. Thirion, O. Grisel, M. Blondel, P. Prettenhofer, R. Weiss, V. Dubourg, J. Vanderplas, A. Passos, D. Cournapeau, M. Brucher, M. Perrot, and D. E. Scikit-learn: Machine Learning in Python . *Journal of Machine Learning Research*, 12:2825–2830, 2011.

[14] A. Shoshani. Multidimensionality in statistical, OLAP, and scientific databases. In *Multidimensional Databases*, pages 46–68. 2003.

[15] H. Søndergaard and P. Sestoft. Referential transparency, definiteness and unfoldability. *Acta Informatica*, 27:505–517, 1990.

[16] P. Vincent, Y. Bengio, N. Chapados, et al. Plearn, 2007. `http://mloss.org/software/view/37/`.

[17] L. Weizsäcker and J. Fürnkranz. Basic instrument for experimental probes in machine learning. Technical Report TUDKE201201, Knowledge Engineering Group, Technische Universität Darmstadt, Apr. 2012.

[18] L. Weizsaecker. peewit, 2012. `http://mloss.org/software/view/254/`.

[19] M. Zitnik and B. Zupan. nimfa a python library for nonnegative matrix factorization, 2012. `http://mloss.org/software/view/397/`.